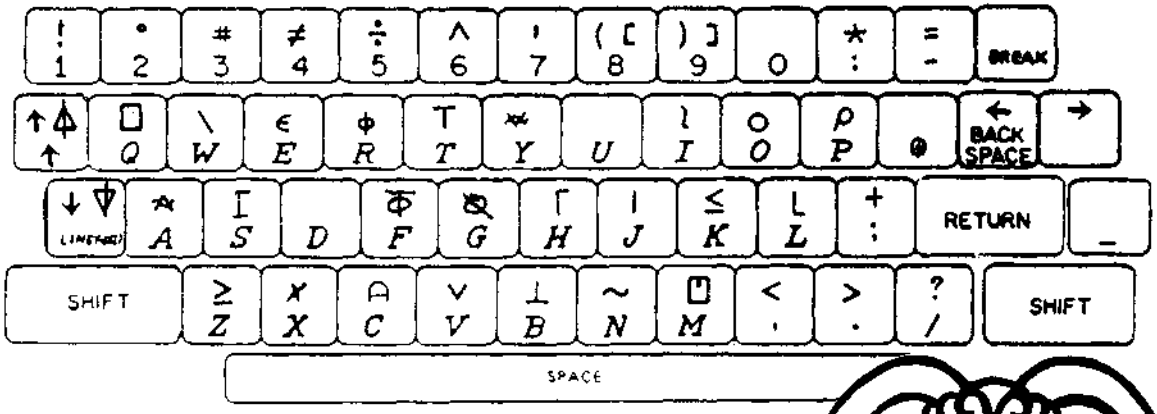
APL 80

by Phelps Gates

APL KEYBOARD



RAMWARE™

TABLE OF CONTENTS

Keyboard Functions 4

Loading 4

Character Set 4

Lessons (Disk) 5

System Commands 5

Function Definition 7

Function Table 12

Function Summary 16

Branching 21

APL80 is a large subset of the APL language, adapted for the TRS-80. These instructions assume that you know APL or have access to a manual which describes it: only the differences and limitations of APL80 are described here.

TRS-80* KEYBOARD FUNCTIONS USING APL 80.

Loading Tape Version

Load APL80 by typing SYSTEM. When computer replies *?, type APL80 and start the tape. When loading is complete, type a slash to enter APL80.

Loading Disk Version

To load APL80 from disk, just type APL.

Character Set

APL80 redefines certain keys, to get characters not normally available. You can type all four arrows by pressing the [SHIFT] key together with the appropriate arrow. Also, you can type an underscore by pressing the [CLEAR] key (APL80 uses this for a negation sign: see below).

The following characters are the same in APL and APL80:

! ' ( ) * : ; = ↑ ↓ ← → + , . ? / > <

For some characters, APL80 uses substitutes:

APL:	÷	●	[	]	‡	^	°	¯	-	⍋ ⍒
APL80:	%	@	(	)	$	&	"	-	_	↑ ↓

For the rest of the APL character set, APL80 uses shifted letters:

APL:	⍟	⍒	⍋	∼	∨	⍲	⍱	○	∣	⎕	⍞	⍳	⍴	∊	⌽	\	⌶	⌈	⌊	⊥	⊤	⍝	⍉	⍕
APL80:	x	z	k	n	v	a	y	o	j	q	m	i	p	e	r	w	s	h	l	b	t	c	g	f

Since an unmodified TRS-80 doesn't distinguish lower-case letters in the video display, APL80 automatically prints a graphics dot at the left of a shifted letter to distinguish it from an ordinary unshifted letter.

The minus signs are a little tricky: APL uses a raised bar for negative numbers and an ordinary minus sign for negation. APL80 moves both of these down one notch: it uses an ordinary minus sign for negative numbers and a lowered minus sign ([CLEAR] key) for negation: APL ¯3-6 becomes APL80 -3_6

Note that round parens are used for indexing (as in BASIC). Also, simple arrows are used for Grade Up/Down: since these are monadic and Take/Drop are dyadic, no ambiguity arises.

The f operator (Disk version only) establishes a format. It must come first in a statement, followed by a vector with an even number of elements, specifying the field size and number of decimals for following numerical output. Note the difference in notation:

APL: 6 3 ⍕ OUTPUT
APL80: f 6 3
OUTPUT

The format continues in effect until (1) a new f statement (2) an error or system command (3) any character output

The d operator (Disk version only) chooses the dimension along which a matrix manipulation is performed (square brackets in APL).

APL: +/[1]MATRIX
APL80: +/1dMATRIX ("plus-reduction on the first dimension")

This applies to reduction, compression, expansion, scan, reversal, and rotation.

Getting Started (Disk version only)

The disk contains five workspaces which explain some of the ways you can use APL. They are:

LESSON1/APL LESSON2/APL
LESSON3/APL LESSON4/APL
LESSON5/APL

To run the first lesson, just type)LOAD LESSON1/APL

System Commands

The following commands correspond to APL, with minor differences:

)OFF)CLEAR)FNS)VARS)SI)ERASE)RESET

In the variable list, any active local or dummy variables are marked with an asterisk.

Tape Operations (tape version only)

)SAVE saves the active workspace to tape
)LOAD loads a workspace from tape
)CHECK verifies a saved workspace (like CLOAD?)

Workspace names are not used. During tape operations, each byte is displayed in the upper right of the screen as it is written or read.

Disk Operations and Disk Version Commands

)SAVE filespec (its previous contents are lost)
)LOAD filespec
)COPY filespec (merges the workspace named by the filespec into the current workspace—duplicate objects replace the corresponding items in the current WS; individual objects cannot be copied)
)DOS returns to DOS READY. You can now execute DOS commands (such as DIR or KILL). To return to APL80 with the workspace intact, type RETURN.
)AUTO expression (puts a latent expression into the workspace: if the workspace is now saved and loaded, this expression will automatically execute, unless you override this feature by holding down the space bar during loading.)
)EXEC executes the latent expression (if any) in the current WS.
)TRON turns on trace: function name and line number are printed just before each statement is executed.
)TROFF turns off trace.

)PS ("print single") makes APL80 print numbers with 6-digit precision. Only output is affected: calculations and variables continue to have 15-digit precision. This will considerably speed up output, especially if format is used.
)PD ("print double") restores 15-digit precision in output.
)KILL filespec will delete a file from the disk. Any file (not just APL-created ones) can be killed.
)RAM enables the Peek, Poke, and Call functions (see below under "RAM Interaction").

The READ, WRITE, RESTORE, UPDATE, and QUERY commands are described below, under "File Handling".

The syntax of functions in APL80 is identical with APL, but the mechanics of function definition differ: the commands)DEF and)EDIT are used to create and to edit (or display) functions.

To create a function, use the)DEF command, followed by the header of the function (Line 0). Examples:

```
)DEF RES←LEFTARG NAME RIGHTARG; LOC1; LOC2
)DEF NILADIC
)DEF VALUE ← MON X
```

The function name must not be in use already, or a DEFN ERROR results.

To change (or display) a function which already exists, use)EDIT, followed by the function name (not the complete header):

```
)EDIT NAME
)EDIT NILADIC
)EDIT MON
```

The)DEF and)EDIT commands put you into definition mode, and APL80 will prompt you for the first unused line of the function (Line 1 for a new function). Type the statement and press [ENTER], and it will ask for the next line.

To leave definition mode, press [BREAK].

To display the function, type)? (and [ENTER]).

To type the function on a printer, type)H (Disk version only).

In the Disk version, you can avoid scrolling in long functions by typing)P followed by a line number; APL80 will display 14 lines, starting with that number. Example:)P21 prints lines 21-34.

To replace a line, type) followed by the line number which you want to replace (and [ENTER]). For example, typing)2 will replace line 2.

To replace a line, type) followed by the line number which you want to replace (and [ENTER]). For example, typing)2 will replace line 2. APL80 immediately deletes the line and prompts you for a replacement. You can replace the header by typing)0. The new header must have the same function name as the old one.

To delete a line, type) and the line number (as above). Then, when APL80 asks you for a replacement, leave definition mode (with [BREAK] or ask for a display with)?.

Whenever you insert or delete a line, APL80 immediately renumbers the lines in sequence. If you ever get confused about the numbers, just type)? for a display of the function with the current line numbers.

As in APL, errors are normally not detected until a function is executed. Exception: syntax errors involving—(such as 3-3) or single-number domain errors (1E99) will be detected during function definition (and cause an exit from definition mode).

In the Disk version, you can edit a function line by typing)E followed by the number of the line which you want to edit. APL80 will display the line at the bottom of the screen, with an edit pointer above

it (in edit mode, the graphics dots appear below the letters, rather than at the left).

To move the edit pointer, press left or right arrows. To insert text at the pointer press I. To delete text at the pointer press D (hold down to repeat). To replace text at the pointer, press R. To cancel changes and start fresh, press A. To complete the edit session and actually change the function line, press [ENTER].

Pressing I or R puts you into insert/replace submode. To exit from insert or replace submode and return to edit mode, press shifted @.

If a line exceeds 64 characters, a length error results. Since lines in function displays replace double quotes with single quotes, you'll need to replace the extra quote before completing the edit of lines which have quotes within quotes. The header cannot be edited.

Limitations

Transcendental functions (including *) are accurate only to 6-digits.

Matrix inverse, lamination, and the diagonal case of dyadic transposition are not implemented. Encode and decode are limited to vector arguments. The arguments of ! must be integers. Hyperbolic functions are not implemented.

Multiple specifications (X←Y←9) must be split into two statements (this also applies to implied multiple specifications, such as %X 9).

A quad can't be typed in response to another quad (quote-quad is OK).

No more than 32 functions can be defined in a workspace, and a function can't have more than 255 lines.

Additional Limitations In Tape Version

Arithmetic in the tape version is limited to 6-digit accuracy.

Scan, transposition, format, inner product, and choice of dimension are not available in the tape version. Catenation applies only to vectors. Arguments of j (residue) may not be negative. The right argument of Take/Drop may not be scalar. Arrays are limited to 5 dimensions (versus 63 in the disk version). The real-time clock is not implemented.

The # and s Operators

The operator # converts numeric arguments to characters and vice versa (like CHR$ and ASC in BASIC). Try, for example, #28 31 150 or #'ABC'.

In addition to the Level II control characters (except 14 and 15) APL80 recongizes:

```
#  3  begin 16-zone print mode
#  5  (Disk only) begin printing hard copy
#  6  (Disk only) randomize workspace (like RANDOM in BASIC)
#  7  begin 8-zone print mode
#  9  (Disk only) stop printing hard copy
# 11  (Disk only) restart random link (?100 will give 77)
# 15  begin 4-zone print mode
# 16  (Disk only) stop real-time clock
# 17  (Disk only) start clock
# 18  (Disk only) reset clock to zero
#255  cancel zone printing
#-1 to #-1023  are equivalent to PRINT @ 1 to PRINT @ 1023
```

Once begin, zone printing continues in effect for all numeric output until cancelled by #255.

If you call for hard copy (with #5) with no printer connected, nothing will happen.

In the disk version, the monadic s operator has these uses:

s0 the time since sign-on (in 25 ms increments)

s1 the "read pointer": the number of the record which will be read by the next READ command (see File Handling)

s2 the "update pointer": next record to be updated

s3 number of records in the most recently accessed file

s4 Line Counter: the number of the line currently executing. To continue execution of a function after pressing [BREAK], type s4.

s5 number of free bytes in the workspace

Miscellaneous

Function and variable names may contain "." and "-": THIS-IS-A-NAME AND.SO.IS.THIS. Watch out: M minus N is "M_N". "M-N" is a reference to a variable named "M-N".

All comparison operators may be used with character arguments (> < $kz=); also Grade Up/Down. But not reduction.

?0 yields a random number between 0 and 1.

[ENTER] always terminates a quoted string. To include carriage returns in a string, use line-feed (unshifted down arrow).

Pressing [BREAK] will terminate function execution, even if the function is awaiting input from a quad or quote-quad.

You can temporarily freeze printed output by holding the space bar down.

The operands of comparison operators must be of the same type ('A'=2 gives a domain error).

The input routine predigests your keyboard input before passing it to the interpreter. This means that statements in function listings and *error messages may differ slightly from what you typed (usually just* in spacing, but try, for example, .00001%0.

The semicolon used to index multi-dimensional ararays does not function as a strong delimiter. This means that parens must be used around compound expressions used as indices: MAT((1+2);2).

Indices need not be integers: APL80 uses the floor, without comment.

The residue function with negative left argument follows the APLSV convention rather than earlier versions of APL.

When syntax errors occur, APL80 sometimes detects them at a different point than APL and behaves differently. Example: if FUNC does not return a value, the statement FUNC+FUNC will result in an error before the function is executed. APL80 also differs from APL in its treatment of duplicate function/label/local variable names.

RAM Interaction Functions

The Disk version includes Peek, Poke, and Call functions. To avoid unfortunate accidents, these functions must first be enabled with the)RAM command.

Monadic u is Peek: the right argument is a RAM location (0 to 65535). The function returns the contents of that location.

Dyadic u is Poke: the left argument is put into the location given by the right argument (example: 65u16000 puts an A onto the screen). It returns a value: the contents of the location after the Poke (the same as the left argument, unless you're Poking ROM!).

Dyadic # is Call: the right argument is put in the A register and the address given by the left argument is CALLed. It returns the value of the A register after the CALL. 51#65 prints "A", for example. #43#0 is equivalent to INKEY$.

File Handling (Disk Version only)

You can move the contents of variables to and from files, using the)WRITE,)READ,)RESTORE,)UPDATE, and)QUERY commands. You can include these commands in a function by spacing before typing the command (this prevents the initial right paren from being interpreted as a function definition command).

)WRITE VARIABLE filespec

The contents of VARIABLE are added to the end of the file. If the file does not exist, it is created, and the contents of VARIABLE become the first record in the file.

)READ VARIABLE filespec

One record (as written by one WRITE command) is read from the file into the variable. Which record gets read is determined by the current value of the "read pointer" (S1). When APL80 first loads, the read pointer is set to 1, and each READ command advances it by 1.

)RESTORE

Resets the read pointer to 1: the next READ command will read the first record in its file.

)QUERY filespec

tells you how many records are in a file, and how big each one is.

A record containing numeric data requires 4+2d+8e bytes, where d is the number of dimensions in the stored variable and e is the total number of elements in it. Character data requires 4+2d+e bytes.

Since each WRITE or READ command requires considerable overhead, it is more economical to write large arrays to disk, rather than many small variables. Files are limited to 127 records, with a total length of less than 65535 bytes.

If the "read pointer" is beyond the last record in a file, an empty vector will be read (you can use this as a test for end of file). Notice that RESTORE doesn't affect writing: the)WRITE command always adds records at the end of the file.

You can change the value of the read pointer (to access records at random) by using s as a dyadic operator with 1 as the second argument. For example: 3 s 1 followed by READ will read the third record. Dyadic s returns the previous value of the pointer — if you don't want it printed, just assign it to a variable: DUMMY 3s1

)UPDATE VARIABLE filespec

writes the contents of VARIABLE to the file, not at the end of the file, but as a replacement for the record pointed to by the "update pointer" (s2). The update pointer is advanced by one. The new record must be the same size as the old one. You can change the value of the update pointer with dyadic s, with 2 as right argument (for example: 4s2 prepares to update the fourth record). RESTORE also resets the update pointer: it's exactly equivalent to 1s1 2. If the update pointer is beyond the end of the file (or if the file doesn't exist), UPDATE is equivalent to WRITE.

The following example shows how file handling works:

```
A←3
B←1 2 3
C←'HELLO'
)WRITE A FILE/DAT          (file created: first object is 3)
)WRITE B FILE/DAT
)WRITE C FILE/DAT
)RESTORE
)READ X FILE/DAT           (X contains 3)
3s1                        prepare to read third record
)READ X FILE/DAT           (X now contains 'HELLO')
)READ X FILE/DAT           (X contains empty vector—
                           endfile)
UP←4 5 6
2s2                        prepare to update second record
)UPDATE UP FILE/DAT
2s1                        now read updated record
)READ Y FILE/DAT           (Y contains 4 5 6)
```

Symbol		Monadic / Dyadic	Name
APL	APL80		
|	.J	M	Absolute Value
+	+	D	Add
∧	&	D	AND
←	←		Assign
→	→		Branch
,	,	D	Catenate
⌈	.H	M	Ceiling
	#		Chr$ / ASC
○	.O	D	Circular
!	!	D	Combinatorial
⍝	.C		Comment
/	/	D	Compress
?	?	D	Deal
⊥	.B	D	Decode
÷	%	D	Divide
↓	↓	D	Drop
⊤	.T	D	Encode
=	=	D	Equal
\	.W	D	Expand
*	*	M	Exponential

Symbol		Monadic / Dyadic	Name
APL	APL80		
!	!	M	Factorial
⌊	.L	M	Floor
⍒	↓	M	Grade Down
⍋	↑	M	Grade Up
>	>	D	Greater Than
≥	.Z	D	Greater or Equal
⍳	.I	M	Index Generator
[ ]	( );	D	Indexing
⍳	.I	D	Index of
.	.	D	Inner Product
:	:		Label
<	<	D	Less Than
≤	.K	D	Less Than or Equal
⍟	∂	D	Log to a Base
⌈	.H	D	Maximum
∈	.E	D	Membership
⌊	.L	D	Minimum
×	.X	D	Multiple
⍲	.A	D	NAND
⍣	@	M	Natural Log
-	-		Negative

Symbol		Monadic / Dyadic	Name
APL	APL80		
⍲	.Y	D	NOR
~	.N	M	Not
≠	$	D	Not equal
∨	.V	D	OR
∘.	".	D	Outer Product
⎕	.Q		Quad
⍞	.M		Quote Quad
?	?	M	Random
,	,	M	Ravel
⌹	%	M	Reciprocal
/	/	M	Reduction
⍴	.P	D	Reshape
|	.J	D	Residue
⌽	.R	M	Reversal
⊖	.R	D	Rotation
⍴	.P	M	Shape
×	.X	M	Sign
⌶	.S	M	System
-	-	D	Subtract
↑	↑	D	Take
⍉	.F	M	Transposition
⌽	.G	M	Transposition
⍕	.G	D	Format

Using APL (Note: Some of these functions are not implemented in the tape version).

The simplest way to use APL80 is just to type an expression (and [ENTER]). APL80 will evaluate it and print the result. For example, try:

```
  2+2
  2%3
3 .X 4 (use shifted X key—a dot will appear)
  7_5 (press [CLEAR] key for underscore)
  2*10
```

APL80 will also operate on groups of several numbers (called 'VECTORS'). For example, try:

```
  2 3 4 + 10 11 12
  5 % 1 2 3
  2 * 0 1 2 3 4 5
```

Use a space to separate the members of a vector. Note what happens if you type: 2 3 + 1 2 3 4. The vectors must be the same length (or one must be a single number).

One slightly tricky thing about APL80 is the way it handles minus-signs. To subtract numbers, use the underscore sign (press [CLEAR] key): 5_3, for example. Use an ordinary minus sign to indicate negative numbers:

```
  -5 + 10 20 30
  10_-30
  3 + -10
```

This takes a little getting used to.

Variables: like BASIC, APL80 can store data in variables: Variable *names may abe any length (up to the length of the input line).* To assign a value to a variable, use the left-arrow (shifted ← key): Try, for example:

```
  Var←1 2 3 4 5
  Var*2
  Var_Var
```

A value error results if you use a variable name which has not been assigned a value.

Commands

APL80 recognizes several system commands, all preceded by a right parenthesis. If you want to consult the disk directory, kill a file, etc., type)DOS

To return to APL80 from DOS, type RETURN.

)ERASE (with a name) will erase a variable or function.

)CLEAR erases everything...you start fresh with an empty workspace.

)SAVE and)LOAD save and load workspaces to and from the disk They must be followed by a valid TRSDOS filespec.

)SI ('State Indicator') lists any functions which have been stopped (by an error or [BREAK]) with the number of the line which was about to execute.)RESET clears the SI: Numerous interrupted functions use up memory and slow down execution (eventually a depth error will occur).

Operators

We've already met + / _ (Watch out for those minus signs!) and .X (Shifted X). Also * (exponentiation). Addition, subtraction, etc., are called 'Dyadic' operations: they involve two numbers, one before and one after the operator. APL80 also has 'Monadic' operators, which use only one argument, after the operator: for example, !5 equals 120 (factorial 5). % can also be monadic (%5 equals .2, the reciprocal of 5). So can + and _ (identity and negative) and .X (sign). Monadic * is the exponential function (E to the Nth). Remember to press [SHIFT] when you type an operator: Otherwise APL80 thinks you are typing a variable.

Order

In APL, there is no hierarchy of operations—it doesn't do multiplication before addition, for example. Expressions are evaluated from the right: For example, 2 .X3 + 10 equals 26...two times (three plus 10). This takes a little getting used to, but it beats worrying about whether 'and' is performed before 'or', etc...in APL, you just evaluate from the right: 2.X3+10_9 is 2.X(3+(10_9)).

If you want to, you can force any order by using parens...(2.X3)+10 is 16.

Index-Gen

The monadic operator .I (Index Generator) produces a vector of numbers: i3 is 1 2 3, for example:

.i0 is a vector of 0 numbers, called an 'empty' vector.

Display

To display, for example, the function 'SQR', type

)EDIT SQR

APL80 will immediately display line 0 (called the header). To see the rest of the function, now type

)?

You can get hard copy by typing

)H (unshifted H)

(Don't do this unless you have a printer ready...) After the display APL80 types the next line number. It's asking you to add a line to the function. If you don't want to, press [BREAK] to leave function mode.

Some functions are like program...you just type their name and they execute. (These are called 'Niladic'). Other functions are like operators, and require arguments: A 'Monadic' function has one argument, a 'Dyadic' function has two.

The variable names in the header of a function are called 'Dummy' variables...they just tell APL80 (and you!) what the function type is, and how many arguments it needs. You can use any names you want (they shouldn't duplicate function names).

Reshape

The reshape operator (.P) constructs an array whose dimensions are given by the first argument, and whose elements are taken from the second argument.

The result will have as many dimensions (up to 64) as there are numbers in the first argument. If the second argument isn't long enough, APL80 goes back to the beginning and starts over.

Reverse

Reverse (.R) is an easy one—note that the reversal occurs on the last dimension (the columns).

Product

The notation A"..XB ('outer product of A and B') means an array generated by multiplying every element of A by every element of B. Any of the operators of lesson 1 may be used: A".+B will add each element, A".=tests to see if they are equal, etc.

The notation A+..XB ('inner product' of two vector means 'multiply each element of A by the corresponding element of B, and sum the resulting vector'. Here too, any of the operators of lesson 1 may be used—441 different combinations are possible.

Quads

The symbol .Q ('QUAD') corresponds to BASIC'INPUT'...try trying: 20.P.Q

You don't have to type a number in response...you can type an expression (2+2), a variable name (it must exist!), or even a function which returns a value...or a character string, in single quotes. .M ('Quote-quad') is similar, except that (1) no prompt is displayed, and (2) what you type is treated as a character string.

Semicolon

You can use ';' to print several things on the same line...this is in addition to the use of ';' for multi-dimensional indexing.

One quirk in the tape version. You can't use ';' in lines which use indexing—APL80 isn't quite smart enough to tell that you're not using it as an index separator.

Tape-Drop

A ↑ B ('A take B') selects A elements from the vector B. If B isn't long enough, it's padded out with zeroes (blanks for a character string). The elements are taken from the start if A is positive, from the end if A is negative.

A ↓ B ('A drop B') is the opposite—it drops A elements from the beginning or end.

Take and drop also work with multidimensional arrays. The left argument must be a vector, with one number for each dimension of the array, telling how many to take (or drop) from that dimension. Take/drop do not work with scalars, in the tape version.

Encode

Encode: (.T) switches from one number system to another. The left argument is a vector containing as many digits as we want in the answer, the right argument is the number which we want to convert: 2 2 2 2 2 2.T21 yields 0 1 0 1 0 1 (21 base 2).

If the left argument is too short, any overflow will be lost. (You can put 0 as the first element in the left vector, in which case any overflow is put into the first element of the result.

The numbers of the left argument need not be the same—to convert 100 inches to yards/feet/inches try 0 3 12.T100.

Decode

Decode (.B) works like this:

Think of the right argument as a vector of digits in a number system whose base is given by the left argument...the result is the value of the number with those digits:

16.B7 15 15 15 gives 32767

For a mixed number system, use a vector on the left, with the same number of elements as the vector on the right (the first may be a dummy). How many seconds are there in 2 weeks, 3 days, 4 hours, 7 minutes and 12 seconds? 1 7 24 60 60.B 2 3 4 7 12

Catenate

A comma adds one vector onto the end of another—this is called 'Catenation'. Try: 1 2 3, 4 5 6

Index

Just as in BASIC, you can use parentheses to extract elements from as array.

If an array has more than one dimension, use a semicolon to separate the dimensions: Mat(2;3). A notation like Mat (;4) means 'all the items in the fourth column'.

You can use indexing on the left of an assignment arrow, to change individual elements of any array.

Rotate

Dyadic .R rotates the right argument left as many places as specified by the left argument: Try:

3.4.i.10

-3.R.i10

Multi-dimensional arrays are rotated along their last dimension (columns).

Grade

↑ (grade up) and ↓ (grade down) are monadic functions which tell the order in which you would need to select the elements of a vector in order to sort them into ascending or descending order. They are usually used in conjunction with indexing.

Shape

Shape (.P) is a monadic function. It yields a result which tells how big an array is—a vector of one number for each dimension: (Don't confuse this with the dyadic .P—Reshape).

.P1 2 3 4, for example, yields 4.

To find the number of dimensions in an array, just use .P twice: .P.P mat. A slightly tricky detail—APL distinguishes between a single number, called a 'Scalar', which has no dimensions (.P3 yields an empty vector), and a vector of 1 element, which has one dimension—you can construct a vector of one element with reshape.

Ravel

The comma can be used as a monadic function to convert an array of any number of dimensions into a vector.

If the argument is a scalar (0 dimensions), it is converted into a vector of 1 element.

Index-of

We met monadic .I (Index Generator) already.

.I can also be used as a dyadic function, to tell where the second argument occurs in the first argument (this is called 'Index of') 9 3 4 6 .I 4 yields 3, because 4 occurs in the third position in the first argument. If the second element doesn't occur in the first, it yields a value one greater than the length of the first argument.

If the second argument occurs more than once, only the first occurrence is found.

Membership

For each element in the left argument, .E checks to see if it is found in the right argument. If it is, it yields 1, if not, 0.

The result has the same number of elements (and dimensions as the left argument).

Try: 1.E1 2 3

Branching Functions

One of the aspects of APL which can be confusing at first is the way it handles branching: the right arrow (→), equivalent to GOTO in BASIC. A non-conditional branch is no problem!—it works just like BASIC. For example:

```
)DEF INFINITELOOP
  1: 'PRESS BREAK TO STOP'
  2: →1
```

Or you can use labels:

```
)DEF SQUAREROOT;X
  1: GETMORE:'ENTER NUMBER'
  2: X←⍞
  3: 'THE SQUARE ROOT OF';X;'IS';X*.5
  4: →GETMORE
```

Note the use of X as a local variable in this function: this allows us to call the function without affecting any value which we may have stored in a variable called X. The shifted q (for "quad") prints a prompt and waits for input (APL⍞).

Conditional branching is a little trickier, since APL doesn't have operators which correspond directly to IF or THEN in BASIC (or FOR...NEXT). Of course, you don't have to use branching as much in APL as in BASIC. Since APL can operate directly on arrays, a single APL line can often do the work of a whole program in BASIC. For example:

```
(⍳8)∘.*⍳8
```

does exactly the same thing as the following BASIC program:

```
10 FOR X=1 TO 8
20 FOR Y=1 TO 8
30 PRINT X↑Y
40 NEXT Y
50 PRINT
60 NEXT X
```

But sometimes you do have to branch conditionally. Consider the following function, which generates Pascal's triangle of binomial coefficients:

```
)DEF TRIANGLE
  1: K←-1
2: ANOTHER:K←K+1
3: 1,(!K)!K
4: →ANOTHER
```

This will print Pascal's triangle, all right, but it won't stop. It just keeps printing lines until the numbers get too big and you get a DOMAIN ERROR. Is there any way to print only the first ten rows? We need to test K and loop back only if K is less than 10, and to do this, we have to know the rules for branching in APL. There are three cases:

1. The line number (or label) exists in the current function. Control simply passes to that line: this is the simplest case, and all the examples so far have been like this:

2. The line number doesn't exist in the function (→0 or →999 or→ -1 or just→). The function terminates: this is equivalent to RETURN in BASIC.

3. The expression to the right of the arrow is an empty vector (i0). No branch occurs; control just passes to the next line.

In the TRIANGLE function, for example, we could change line 4 to:

```
4:  ANOTHER x K < 10
```

Now if K is less than 10, the logical expression K<10 will have the value 1 (true), and the expression in line 4 will have the value ANOTHER x 1, which equals ANOTHER. On the other hand, if K is not less than 10, the expression K < 10 has the value 0 (false), and the function branches to ANOTHER x 0, which equals 0, and so the function terminates, by rule 2.

We could also write line 4 as:

```
4:  ANOTHER x i K < 10
```

If K is less than 10, this branches to ANOTHER x i 1, and since i 1 equals 1, this is equivalent to →ANOTHER. If K is not less than 10, we get

```
    ANOTHER x i 0
```

and since multiplying an empty vector by anything still gives an empty vector, this is equivalent to → i 0. No branch occurs (rule 3), but since there aren't any more lines in the function, execution terminates anyway.

This works fine mathematically, but it doesn't make your programs any easier to read, and it's a nuisance to have to figure all this out every time you write a function. One solution would be to define a function to figure it out for us. For example:

```
)DEF BRANCH←LABEL IF CONDITION
  1:BRANCH←LABEL x i CONDITION
```

Now we can write line 4 of the TRIANGLE function as

```
4:  ANOTHER IF K<10
```

which makes a little more sense. The function IF computes the proper destination for the branch, depending on the value of K. Note that we don't have to call it IF; you could call the function PROVIDED.THAT or SI or whatever. It's a dyadic function, requiring both a left operand (the label) and a right operand (the condition), and it returns a value (the appropriate destination).

You can also do it backward, and define a function called UNLESS:

```
)DEF BRANCH←LABEL UNLESS CONDITION
  1:BRANCH←LABEL x i n CONDITION
```

and rewrite the TRIANGLE function as:

```
4:  ANOTHER UNLESS K>9
```

Instead of using multiplication, you can define the IF function using the compression operator:

```
1:BRANCH←CONDITION/LABEL
```

MODIFYING DISK VERSION

To modify APL 80 for lower case, different printers, or RAM interactive functions, load the workspace called CUSTOM/APL and invoke the function to make the change.

To save changes in APL 80

1.) Goto DOS
2.) Type DEBUG and press ENTER
3.) You will automatically go into DEBUG
4.) Make your changes in memory
5.) Type G402D and press ENTER to return to DOS
6.) Type DEBUG (OFF) and press ENTER
7.) Type TAPEDISK
8.) From Tapedisk, type:
    F APL/CMD:0 (6000 9AFA 71A3)
    and press ENTER
9.) Type E and press ENTER
10.)You now have a new version of APL80 with the name APL instead of APL80

)SAVE INST/APL

APL 80

APL, the elegant computer language, comes with all these features:

Self-teaching lessons **15 digit accuracy**
Over 60 functions **11 control characters**
Random and sequential file handling

Easy to learn, easy to use and very powerful. The cassette version has no lessons, 6 digit accuracy, fewer functions and no file handling.

6 SOUTH ST., MILFORD, N.H. 03055 (603) 673-5144